CHAPTER 5

CATALOG OF ATOMIC CHANGES IN JAVA

This chapter describes the various atomic changes possible in a Java program. The format in which each atomic change is documented is the same as the one described in Chapter 4.

5.1 ATOMIC CHANGES

An investigation of Java language reveals 58 atomic changes. It will be seen that some atomic changes are common to both C++ and Java, whereas others are distinct. The commonality is explained by the fact that Java design has been influenced by C++, whereas the difference exposes fundamental divergence in language design philosophy and goal. For example, accessibility qualification using keywords private, public and protected is borrowed from C++, whereas synchronized keyword arises out of the need to support multithreading. Similarly, the absence of template mechanism in Java appears to hinge on the goal to keep the language simple.

A summary of the atomic changes detailed in this chapter may be found in Appendix 4.

5.1.1 Add comments

5.1.1.1 Description
This corresponds to adding comments within a class. The assumption is that the new comment does not invalidate code.

5.1.1.2 Example

Before change:
```java
class Test {
    int a;
    public void print() {
        System.out.println("Test.a =" + a);
    }
}
```
After change:
class Test {
    int a;
    // Print the instance variable for debug purposes
    public: void print() {
        System.out.println("Test.a = " + a);
    }
}

5.1.3 Equivalence
This atomic change preserves structural equivalence. Hence there is no need to retest the changed program.

5.1.4 Reason for change
Programmers are expected to liberally comment their source code. Inadequate level of commenting may be pointed out by code reviewers, forcing the programmer to add more comments at a later point of time.

5.1.2 Delete comments

5.1.2.1 Description
This corresponds to deleting comments from a class. The assumption is that this change does not activate new code.

5.1.2.2 Example
Before change:
class Test {
    int a;
    // Print the instance variable for debug purposes
    public: void print() {
        System.out.println("Test.a = " + a);
    }
}

After change:
class Test {
    int a;
public void print() {
    System.out.println("Test.a = " + a);
}

5.1.2.3 Equivalence
This atomic change preserves structural equivalence. Hence there is no need to retest the changed program.

5.1.2.4 Reason for change
Trivial comments that repeat the actual code statement are rarely useful. Reviewers might recommend deleting such comments.

5.1.3 Change member accessibility level

5.1.3.1 Description
Change the accessibility level of an element of a class. Java supports four levels of accessibility: private, protected, public, and friendly. Private elements are visible only to methods of the class; public elements are visible anywhere the class itself is visible; protected members are visible to methods of the class, methods of derived classes, and to classes defined in the same package; friendly elements are visible to classes in the same package as well as to methods of the class.

5.1.3.2 Example

Before change:

class A {
    public int a;
    private float f;
    public void ff() { f += a; }
}

After change:

class A {
    protected int a;
    private float f;
    public void ff() { f += a; }
}
5.1.3.3 Equivalence

This atomic change may or may not preserve equivalence. The different cases are shown in Table 5.1.

<table>
<thead>
<tr>
<th>Element modified</th>
<th>Equivalence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance variable</td>
<td>T-eq</td>
</tr>
<tr>
<td>Class variable</td>
<td>T-eq</td>
</tr>
<tr>
<td>Static inner class/interface</td>
<td>T-eq</td>
</tr>
<tr>
<td>Nonstatic inner class/interface</td>
<td>T-eq</td>
</tr>
<tr>
<td>Instance method</td>
<td>No</td>
</tr>
<tr>
<td>Class method</td>
<td>T-eq</td>
</tr>
</tbody>
</table>

Table 5.1 Access Level Change

As shown in this table, when the access level of an instance method is changed, it may alter the behavior of the class and hence retesting is needed. In all other cases, there is no behavior change, implying no retesting. Consider the following example:

**Before change:**
```java
class Base {
    public void f() { g(); }
    public void g() {
    }
}
class Derived extends Base {
    public void g() {
    }
}
class JavaTest {
    public static void main(String args[]) {
        Derived d = new Derived();
        d.f(); // Invokes Derived.g()
    }
}
```

**After change:**
```java
class Base {
    public void f() { g(); }
    private void g() {
    }
}
```
class Derived extends Base {
    public void g() {
    }
}
class JavaTest {
    public static void main(String args[]) {
        Derived d = new Derived();
        d.f(); // Invokes Base.g()
    }
}

The behavior of Base.f() has changed in the modified version, requiring retesting.

5.1.3.4 Reason for change
It has been observed that during initial code development, programmers are generally lax when it comes to choosing the most appropriate accessibility qualifier. Many coding guidelines recommend, for example, making data members private. A programmer who follows this rule automatically makes data members private, but may realize that some derived class requires to have access to this element at a later time, and changes the accessibility to protected. Similarly, most of the methods are qualified as public (to start with), but some are eventually changed to private if they are used for internal code factoring.

5.1.4 Add instance variable

5.1.4.1 Description
Introduce a new instance variable in the class.

5.1.4.2 Example

Before change:

class AddInstVar {
    private int a = 6;
    public void f() { a++; }
}
After change:

```java
class AddInstVar {
    private int a = 6;
    protected int val = 0;
    public void f() { a++; }
}
```

5.1.4.3 Equivalence

This change may or may not preserve T-equivalence. If defining the instance variable alters the LFC of any of the existing methods, then T-equivalence is not preserved. Otherwise, T-equivalence is preserved. Both cases are illustrated below.

Case 1 (LFC Unaltered):

Before change:

```java
class AddInstVar {
    private int a = 6;
    public void f() { a++; }
}
```

After change:

```java
class AddInstVar {
    private int a = 6;
    protected int val = 0;
    public void f() { a++; }
}
```

In this case, the newly introduced instance variable is not used by any method, so the LFC of each of the methods remains the same after change. Therefore, the class after change is T-equivalent to the original one.

Case 2 (LFC altered):

Before change:

```java
class Base {
    protected int val = 23;
}
```
class Derived extends Base {
    private int a = 6;
    public void f() {
        a++;
        val--;
    }
}

After change:

class Base {
    protected int val = 23;
}
class Derived extends Base {
    private int a = 6;
    protected int val = 23;
    public void f() {
        a++;
        val--;
    }
}

Clearly, the behavior of method f() in Derived is changed since it binds to the new instance variable and not to the base element!

5.1.4.4 Reason for Change
This change is typically not complete by itself. The class designer usually follows this change with other changes such as changing method body. Since instance variables of a class preserve object state across time, the primary reason for defining a new instance variable in a class across versions is to model the enhanced state.

5.1.5 Change interface declaration order

5.1.5.1 Description
Change the order of interfaces in the declaration list.
5.1.5.2 Example

**Before Change:**

class A implements K, I, J {
   // Body
}

**After Change:**

class A implements I, J, K {
   // Body
}

5.1.5.3 Equivalence

This change preserves S-equivalence. Hence there is no need to perform retest after this change.

5.1.5.4 Reason for change

This is a very trivial change. The only reason why a developer might do this is to follow some coding guideline, for example, one that requires the list of interfaces to be in a specific lexicographic order.

5.1.6 Add instance method

5.1.6.1 Description

An instance method is defined in the class.

5.1.6.2 Example

**Before change:**

class A {
   public void ff() {}
}

**After change:**

class A {
   public void ff() {}
   void gg() {}
}
5.1.6.3 Equivalence
This atomic change does not preserve T-equivalence. Retesting is needed.

5.1.6.4 Reason for change
Adding new methods in a class may occur as part of class functionality enhancement.

5.1.7 Delete instance method

5.1.7.1 Description
An instance method is deleted from the class.

5.1.7.2 Example

Before change:
```java
class A {
    public void ff() { }
    void gg() {}
}
```

After change:
```java
class A {
    public void ff() { }
}
```

5.1.7.3 Equivalence
This atomic change does not preserve T-equivalence. Retesting is needed.

5.1.7.4 Reason for change
Deleting one or more instance methods is a major code change and may occur as part of code restructuring.

5.1.8 Add class method

5.1.8.1 Description
A new static method is introduced in the class.
5.1.8.2 Example

Before change:
class A {
    public void ff() {
    }
}

After change:
class A {
    public void ff() {
    static void gg() {}
    }
}

5.1.8.3 Equivalence
This atomic change does not preserve T-equivalence. Retest is required.

5.1.8.4 Reason for change
Adding new methods in a class may occur as part of class functionality enhancement.

5.1.9 Delete class method

5.1.9.1 Description
An existing static method is deleted from the class.

5.1.9.2 Example

Before change:
class A {
    public void ff() {
    static void gg() {}
    }
}

After change:
class A {
    public void ff() {
    }
}

5.1.9.3 Equivalence
This atomic change does not preserve T-equivalence. Retest is required.
5.1.9.4 **Reason for change**
Deleting one or more static methods is a major code change and may occur as part of code restructuring.

5.1.10 **Make a method synchronized**

5.1.10.1 **Description**
Java allows methods of a class to be qualified as *synchronized* to render the class thread safe. This atomic change corresponds to qualifying a method as synchronized in a new version whereas in the original version it was not.

5.1.10.2 **Example**

**Before change:**
```java
class A {
    private int val = 0;
    public void setValue(int v) {
        val = v;
    }
    public int getValue() {
        return val;
    }
}
```

**After change:**
```java
class A {
    private int val = 0;
    synchronized public void setValue(int v) {
        val = v;
    }
    synchronized public int getValue() {
        return val;
    }
}
```

5.1.10.3 **Equivalence**
This atomic change preserves T-equivalence. If the class was correct before this atomic change was introduced, it must be correct
after the change since the behavior does not change. As a result, there is no need to retest the class after modification.

5.1.10.4 **Reason for change**

This is a very important change to a class method. There are good reasons for designing a thread-safe class, i.e., a class that guarantees correct semantics in the presence of concurrent threads of execution. Making all methods synchronized is one common way of making a class thread safe (this is not sufficient, however). Hence the designer of a class may, as part of enhancements for a future release, may synchronize the class methods.

5.1.11 **Delete instance variable**

5.1.11.1 **Description**

An instance variable of the class is deleted in the modified version.

5.1.11.2 **Example**

**Before change:**

```java
class DellInstVar {
    private int a = 6;
    protected int val = 0;
    public void f() { a++; }
}
```

**After change:**

```java
class DellInstVar {
    private int a = 6;
    public void f() { a++; }
}
```

5.1.11.3 **Equivalence**

This change may or may not preserve T-equivalence. If deleting the instance variable alters the LFC of any of the existing methods, then T-equivalence is not preserved. Otherwise, T-equivalence is preserved. Both cases are illustrated below.
Case 1 (LFC Unaltered):

**Before change:**

class DelInstVar {
    private int a = 6;
    protected int val = 0;
    public void f() { a++; }
}

**After change:**

class DelInstVar {
    private int a = 6;
    public void f() { a++; }
}

In this case, the deleted instance variable is not used by any method, so the LFC of each of the methods remains the same after change. Therefore, the class after change is T-equivalent to the original one.

Case 2 (LFC altered):

**Before change:**

class Base {
    protected int val = 23;
}
class Derived extends Base {
    private int a = 6;
    protected int val = 23;
    public void f() {
        a++;
        val--;
    }
}

**After change:**

class Base {
    protected int val = 23;
}
class Derived extends Base {
    private int a = 6;
    public void f() {
        a++;
        val--; 
    }
}

Clearly, the behavior of method f() in Derived is changed since it binds to the base element.

5.1.11.4 Reason for change
This change is typically not complete by itself. The class designer usually follows this change with other changes such as changing method body.

5.1.12 Make a method nonsynchronized

5.1.12.1 Description
Turn a method that was originally synchronized into nonsynchronized.

5.1.12.2 Example
Before change:

class A {
    private int val = 0;
    synchronized public void setValue(int v) {
        val = v;
    }
    synchronized public int getValue() {
        return val;
    }
    synchronized public int getValueNorm() {
        return val + val;
    }
}

After change:

class A {
    private int val = 0;
public void setValue(int v) {
    val = v;
}
public int getValue() { return val; }
public int getValueNorm() {
    return val + val;
}

5.1.12.3 Equivalence
This atomic change does not guarantee T-equivalence. The modified class needs to be retested since it is not thread-safe. In the special case where the application is single threaded, T-equivalence is preserved.

5.1.12.4 Reason for change
The context for this atomic change has to be understood in terms of multithreading. When methods of a class are synchronized, the responsiveness of certain parts of the application may deteriorate. This is due to the fact that the Java Virtual Machine blocks multiple calls to a synchronized method on the same object by different threads. To understand this, consider the class given above (before change). If two different threads invoke getValue() and getValueNorm() respectively on the same A object, only one will be allowed to execute, to ensure atomicity of the operation. However, it is easy to see that there is no harm in allowing the two methods to execute concurrently as they are both read methods. To overcome this inherent weakness of the JVM, the class designer might remove the synchronize qualifier from all methods, leaving it to the client of the class to handle it correctly.
5.1.13 Make a method final

5.1.13.1 Description
A nonfinal method of a class is made final. This atomic change is feasible only when the method has not been overridden in any derived class.

5.1.13.2 Example
Before change:

```java
class A {
    public void ff() {/* ... */ }
    public void gg() {/* ... */ }
}
class B extends A {
    public void gg() {/* ... */ }
}
```

After change:

```java
class A {
    public final void ff() {/* ... */ }
    public void gg() {/* ... */ }
}
class B extends A {
    public void gg() {/* ... */ }
}
```

Note that A.gg() cannot be made final since it is overridden by B.gg().

5.1.13.3 Equivalence
Since this method is not overridden in any derived class, making it final has no behavior impact. In other words, this atomic change preserves T-eq.

5.1.13.4 Reason for change
The reason for declaring a method final is to prohibit overriding so that the method behavior remains unchanged even in the context of future derived classes. As an example, consider the following:
class Employee {
    public final String getName() {
        return name;
    }
    public void describe() { /* ... */ }
    // ...
    private String name;
}

class Manager extends Employee {
    public double getAllowance() { /* ... */ }
    public void describe() { /* ... */ }
    // ...
}

In this example, Employee.getName() is made final since the designer wants to ensure fixed meaning for its behavior. That is, even for objects of classes derived from Employee, the behavior should be as dictated by Employee.

5.1.14 Make class public

5.1.14.1 Description
A nonpublic class is made public. For this atomic change to be feasible, there are two requirements:

• the name of the source file should be the same as the class name
• there should be no other public class in the same source file

5.1.14.2 Example
Before change:

class X {
    // ...
}

After change:

public class X {
    // ...
}
5.1.14.3 **Equivalence**
Making a class public does not change the behavior of the class. It only affects its accessibility. Hence this atomic change preserves T-equivalence.

5.1.14.4 **Reason for change**
A class initially meant for limited use within a package is likely to be nonpublic. In the next version, realizing the wider utility of the class (across packages), the designer might make it public, thus allowing it to be importable.

5.1.15 **Add direct initialization for instance variable**

5.1.15.1 **Description**
For an instance variable, define a direct initialization expression. This expression is evaluated each time an instance of the class is created and the instance variable is initialized with that value.

5.1.15.2 **Example**

*Before change:*
```java
class A {
    private int val;
    // ...
}
```

*After change:*
```java
class A {
    private int val = 12;
    // ...
}
```

5.1.15.3 **Equivalence**
There is, in general, no way to determine if the direct initialization alters the behavior of the class. Therefore, the conclusion is that this atomic change does not necessarily preserve equivalence, and hence retesting is necessary. To give an (extreme) example of when retesting is not necessary, consider:
Before change:

class A {
    private int val;
    public A() {
        val = 12;
    }
    // ..
}

After change:

class A {
    private int val = 1234;
    public A() {
        val = 12;
    }
    // ..
}

In the changed version, the effect of direct initialization is nullified by the assignment made in the constructor! Following is an example requiring retest:

Before change:

class A {
    private double dval;
    private int val;
    public A() {
        dval = 12.4567;
    }
    // ..
}

After change:

class A {
    private double dval;
    private int val = 1234;
    public A() {
        dval = 12.4567;
    }
}
In this case, the initial value of instance variable \texttt{val} is different in the two cases, and hence retesting is called for.

5.1.15.4 \textbf{Reason for change}

One good (and frequently applied) context to introduce this atomic change is when an instance variable is initialized with a fixed value irrespective of how the object is instantiated. In all the following cases, this atomic change is recommended:

Case 1:

```java
class A {
    private int val;
    private double dval;
    public A() {
        val = 12;
        dval = 12.345;
    }
    // other nonconstructor methods and fields
}
```

Case 2:

```java
class A {
    private int val;
    private double dval;
    public A() {
        val = 12;
        dval = 12.345;
    }
    public A(double dv) {
        val = 12;
        dval = dv;
    }
    // other nonconstructor methods and fields
}
```
Case 3:

```java
class A {
    private int val;
    private double dval;
    {
        val = 12;
    }
    public A() { dval = 12.345; }
    public A(double dv) { dval = dv; }
    // other nonconstructor methods and fields
}
```

In all the above three cases, it is better to rewrite the class using direct initialization as under:

```java
class A {
    private int val = 12;
    private double dval;
    public A() { dval = 12.345; }
    public A(double dv) { dval = dv; }
    // other nonconstructor methods and fields
}
```

5.1.16 Remove direct initialization for instance variable

5.1.16.1 Description
If an instance variable is initialized at the point of declaration, remove this.

5.1.16.2 Example

Before change:

```java
class A {
    int count = 1;
    public void ff() { /* ... */ }
}
```

After change:

```java
class A {
    int count;
```
public A(int c) {
    count = c;
}
public void ff() { /* ... */ }

5.1.16.3 Equivalence
This change, in general, does not guarantee equivalence. Retesting is therefore required.

5.1.16.4 Reason for change
One reason for removing direct initialization is that the default value is acceptable (for example, all integers are initialized automatically with zero). The other reason is that initial value needs to be computed dynamically.

5.1.17 Add direct initialization for class variable

5.1.17.1 Description
For a class variable, define a direct initialization expression. This expression is evaluated exactly once for the class, when the class is first loaded into the virtual machine and the class variable is initialized with that value.

5.1.17.2 Example
Before change:
class A {
    private static int val;
    // ...
}

After change:
class A {
    private int static val = 12;
    // ...
}

5.1.17.3 Equivalence
There is, in general, no way to determine if the direct initialization alters the behavior of the class. Therefore, the conclusion is that
this atomic change does not necessarily preserve equivalence, and hence retesting is necessary. To give an (extreme) example of when retesting is not necessary, consider:

**Before change:**

class A {
    private static int val;
    static {
        val = 12;
    }
    // ..
}

**After change:**

class A {
    private static int val = 1234;
    static {
        val = 12;
    }
    // ..
}

In the changed version, the effect of direct initialization is nullified by the assignment made in the static initializer! Following is an example requiring retest:

**Before change:**

class A {
    private static double dval;
    private static int val;
    static {
        dval = 12.4567;
    }
    // ..
}

**After change:**

class A {
    private static double dval;
private static int val = 1234;
static {
    dval = 12.4567;
}

In this case, the initial value of class variable val is different in the two cases, and hence retesting is called for.

5.1.17.4 Reason for change
This atomic change is applicable if the class variable is to be initialized with a nonzero value, in the form of an expression that does not raise an exception.

5.1.18 Remove direct initialization for class variable

5.1.18.1 Description
If a class variable is initialized at the point of declaration, remove this.

5.1.18.2 Example
Before change:

```java
class A {
    static int count = 1;
    public void ff() { /* ... */ }
}
```

After change:

```java
class A {
    static int count;
    static {
        count = 1;
    }
    public void ff() { /* ... */ }
}
```

5.1.18.3 Equivalence
This change, in general, does not guarantee equivalence. Retesting is therefore required.
5.1.18.4 **Reason for change**

One reason for removing direct initialization is that the default value is acceptable (for example, all integers are initialized automatically with zero). The other reason is that the initialization expression may throw an exception, in which case, it has to be within a static initializer. This is shown in the following example:

```java
class A {
    public static int value;
    static {
        try {
            value = ff();
        } catch (MyException m) {
            value = 0;
        }
    }
    private int ff() throws MyException {
        // ... body
    }
}
```

The above static initialization cannot be done outside the static block.

5.1.19 **Change instance variable type**

5.1.19.1 **Description**

Change the type of an instance variable.

5.1.19.2 **Example**

**Before change:**

```java
class Person {
    private int age;
    // other fields and methods
}
```

**After change:**

```java
class Person {
    private short age;
}
```
5.1.19.3 Equivalence
In general, there is no guarantee that the behavior will be the same after this atomic change. Hence this change does not necessarily preserve equivalence and retesting is called for.

5.1.19.4 Reason for change
The choice of type for an instance variable is based on the set of values (domain) the variable needs to represent. For example, in the case of the Person class shown above, the designer feels 32 bits are not needed for representing age. So he chooses to use short (which is only 16 bits).

5.1.20 Make an instance variable transient

5.1.20.1 Description
An instance variable is qualified as transient. Transient instance variables are not serialized or deserialized.

5.1.20.2 Example

Before change:
```java
class Employee {
    private short age;
    private double salary;
    // ...
}
```

After change:
```java
class Employee {
    private short age;
    private transient double salary;
    // ...
}
```

5.1.20.3 Equivalence
The transient keyword has no impact on the logical behavior of the class. The only change occurs in terms of serialization/
deserialization. If objects of the class reside only in memory, there
is no need to retest the class. But if objects are serialized and
deserialized, then the class has to be retested after this
modification. Thus, this atomic change may or may not preserve T-
equivalence.

5.1.20.4 Reason for change
This change is relevant only in the context of object serialization/
deserialization.

5.1.21 Make an instance variable nontransient

5.1.21.1 Description
Remove the transient qualifier from an instance variable.

5.1.21.2 Example

Before change:

class Employee {
    private short age;
    private transient double salary;
    // ...
}

After change:

class Employee {
    private short age;
    private double salary;
    // ...
}

5.1.21.3 Equivalence
The transient keyword has no impact on the logical behavior of the
class. The only change occurs in terms of serialization/
deserialization. If objects of the class reside only in memory, there
is no need to retest the class. But if objects are serialized and
deserialized, then the class has to be retested after this
modification. Thus, this atomic change may or may not preserve equivalence.

5.1.21.4 Reason for change
This change is relevant only in the context of object serialization/deserialization.

5.1.22 Change method body

5.1.22.1 Description
Change the body of a method.

5.1.22.2 Example

**Before change:**
```java
class A {
    private int val = 9;
    public int getVal() { return val; }
}
```

**After change:**
```java
class A {
    private int val = 9;
    public int getVal() {
        return val + 2;
    }
}
```

5.1.22.3 Equivalence
In general, it is not possible to determine whether two pieces of code perform the same computation. Since equivalence may not be preserved, the conclusion is that retest is necessary.

5.1.22.4 Reason for change
There are several reasons for a method body to change. Some are correcting a bug, speeding up performance, functionality enhancement, and code factoring.
5.1.23 Change order of elements in throws list

5.1.23.1 Description
A method is required to specify a list of exceptions that it directly or indirectly throws. This atomic change is about changing the order of elements in the throws list.

5.1.23.2 Example

**Before change:**
```java
class Sample {
    public void ff() throws Q, P, R {
        // Body
    }
}
```

**After change:**
```java
class Sample {
    public void ff() throws P, Q, R {
        // Body
    }
}
```

5.1.23.3 Equivalence
This atomic change preserves S-equivalence. Hence there is no need to retest the class after modification.

5.1.23.4 Reason for change
This is a pure cosmetic change, perhaps necessitated due to coding conventions, e.g., the throws list must be in lexicographically ascending order (for readability).

5.1.24 Make class nonpublic

5.1.24.1 Description
A class originally public is made nonpublic.
5.1.24.2 Example

**Before change:**
```
public class ListNode {
    // details
}
```

**After change:**
```
class ListNode {
    // details
}
```

5.1.24.3 Equivalence
Since the keyword applied to a class only affects accessibility, not its semantics, there is no need to retest the class. Preserves T-equivalence.

5.1.24.4 Reason for change
A public class signifies wider use (outside of the package it belongs to). As part of redesigning the classes in the next version, some class might get restricted to its own package. As in the example above, ListNode was originally thought to be useful to several packages, but in the next version was made nonpublic due to its limited use.

5.1.25 **Change method argument order**

5.1.25.1 Description
Change the order in which method arguments are declared.

5.1.25.2 Example

**Before change:**
```
class Sample {
    public void ff(int a, String s) {
        // Body
    }
}
```
After change:

class Sample {
    public void ff(String s, int a) {
        // Body
    }
}

5.1.25.3 Equivalence

The class has to be retested since the behavior may change after this modification. The following is one such example.

Before change:

class Base {
    public void ff(int a, String s) {
    }
}
class Derived extends Base {
    public void ff(String s, int a) {
    }
    public void gg() {
        ff(12, "Hello"); // This invokes Base::ff()
    }
}

After change:

class Base {
    public void ff(int a, String s) {
    }
}
class Derived extends Base {
    public void ff(String s, int a) {
    }
    public void gg() {
        ff(12, "Hello"); // This invokes Derived::ff()
    }
}

It can be seen from this example that the behavior of Derived::gg() has changed! This is due to the fact Java allows methods across scopes (e.g. in different classes in a hierarchy) to overload one
another. The author feels this is a dangerous feature of Java – something that C++ does not have!

5.1.25.4 **Reason for change**
There is no convincing reason for introducing this atomic change, especially when it results in subtle (sometimes difficult to detect) behavior changes.

5.1.26 **Make class final**

5.1.26.1 **Description**
A nonfinal class is qualified as final. This atomic change is feasible only if the class has no descendent classes.

5.1.26.2 **Example**

**Before change:**
```java
class Sample {
    // ...
}
```

**After change:**
```java
final class Sample {
    // ...
}
```

5.1.26.3 **Equivalence**
Making a class final has no impact on its behavior, but only prevents creation of classes derived from it. Hence this atomic change preserves T-equivalence.

5.1.26.4 **Reason for change**
If the designer feels that the class should permit no derivations from it, the class can be made final. Note that the final class may have base classes.

5.1.27 **Make class nonfinal**

5.1.27.1 **Description**
A final class is made nonfinal by removing the final qualifier.
5.1.27.2 Example

Before change:

```java
final class Sample {
    // ...
}
```

After change:

```java
class Sample {
    // ...
}
```

5.1.27.3 Equivalence

Whether a class is final or not has no bearing on its logical behavior. Hence this preserves T-equivalence.

5.1.27.4 Reason for change

The designer may realize, as part of the new version, that there is a need to derive from this class. Since a final keyword precludes derivation, he may remove the qualifier to make the class derivable.

5.1.28 Add a base class

5.1.28.1 Description

Derive the class from another class. There are two possibilities:

- the class does not have any explicit base class (in that case, `Object` is the implicit base class)
- the class is already derived from another base class

5.1.28.2 Example

Case 1: There is no explicit base class

Before change:

```java
class Sample {
    public void ff() {
        // ...
    }
}
```
After change:

class Base {
    // ...
}

class Sample extends Base {
    public void ff() {
        // ...
    }
}

Case 2: There is already a base class

Before change:

class Base {
    protected void gg() {
        // ...
    }
}

class Sample extends Base {
    public void ff() { gg(); }
}

After change:

class Base {
    protected void gg() {
        // ...
    }
}

class NewBase extends Base {
    protected void gg() {
        // ...
    }
}

class Sample extends NewBase {
    public void ff() { gg(); }
}

5.1.28.3 Equivalence

This change may or may not preserve T-equivalence. In the first case above, the behavior of Sample does not change after adding a
base. However, in the second case, the behavior has changed. In general, therefore, retesting is necessary.

5.1.28.4 Reason for change
This atomic change is likely when the designer performs code factoring resulting in changes in the class hierarchy. The primary reason to add a base class is to inherit behavior provided by an existing class.

5.1.29 Delete base class

5.1.29.1 Description
Delete the base class of the current class. This makes Object as the base class since every Java class has Object as the root of the hierarchy. For this to be a valid atomic change, no specific element (field or method) of the deleted base class should be used within this class. Additionally, the base elements should not have been accessed via the derived class.

5.1.29.2 Example

Before change:
```java
class Base {
    public void ff() { /* ... */ }
}
class Derived extends Base {
    public int getValue() {
        return val += 56;
    }
    private int val = 9;
}
```

After change:
```java
class Base {
    public void ff() { /* ... */ }
}
```
```java
class Derived {
    public int getValue() {
        return val += 56;
    }
    private int val = 9;
}
```

5.1.29.3 Equivalence
This change preserves T-equivalence.

5.1.29.4 Reason for change
This atomic change is likely when the designer performs code factoring resulting in changes in the class hierarchy.

5.1.30 Make class abstract

5.1.30.1 Description
A class is tagged with the abstract qualifier. This atomic change pertains to the case where no method of the class is abstract. The other case when a class needs to be qualified as abstract is when at least one of its methods is abstract. This is a valid atomic change only when the class has not been instantiated anywhere in the code.

5.1.30.2 Example

Before change:
```java
class Math {
    public static int min(int a, int b) { /* ... */ }
    public static int max(int a, int b) { /* ... */ }
}
```

After change:
```java
abstract class Math {
    public static int min(int a, int b) { /* ... */ }
    public static int max(int a, int b) { /* ... */ }
}
```

5.1.30.3 Equivalence
Making a class abstract does not change its behavior. Hence retesting is not necessary. It preserves T-equivalence.
5.1.30.4 **Reason for change**

One reason for making a class abstract is when the class has only static elements and hence instantiation would not make sense.

5.1.31 **Change method signature**

5.1.31.1 **Description**

Change the signature of a method. Return type is included as part of this atomic change.

5.1.31.2 **Example**

*Before change:*

```java
class Sample {
    public void f(int a) { /* ... */ }
}
```

*After change:*

```java
class Sample {
    public void f(long a) {/*... */}
}
```

5.1.31.3 **Equivalence**

This atomic change does not preserve equivalence. Retesting is needed.

5.1.31.4 **Reason for change**

Argument type promotions as shown in the example, and other signature changes, are sometimes done as part of code review.

5.1.32 **Add class variable**

5.1.32.1 **Description**

Add a new class (static) variable to the class.

5.1.32.2 **Example**

*Before change:*

```java
class AddClassVar {
    private int a = 6;
}
```
public void f() { a++; }
}

After change:
class AddClassVar {
    private int a = 6;
    protected static int val = 0;
    public void f() { a++; }
}

5.1.32.3 Equivalence
This change may or may not preserve T-equivalence. If defining the class variable alters the LFC of any of the existing methods, then T-equivalence is not preserved. Otherwise, T-equivalence is preserved. Both cases are illustrated below.

Case 1 (LFC Unaltered):

Before change:
class AddClassVar {
    private int a = 6;
    public void f() { a++; }
}

After change:
class AddClassVar {
    private int a = 6;
    protected static int val = 0;
    public void f() { a++; }
}

In this case, the newly introduced instance variable is not used by any method, so the LFC of each of the methods remains the same after change. Therefore, the class after change is T-equivalent to the original one.
Case 2 (LFC altered):

**Before change:**

```java
class Base {
    protected int val = 23;
}
class Derived extends Base {
    private int a = 6;
    public void f() { a++; val--; }
}
```

**After change:**

```java
class Base {
    protected int val = 23;
}
class Derived extends Base {
    private int a = 6;
    protected static int val = 23;
    public void f() { a++; val--; }
}
```

Clearly, the behavior of method `f()` in Derived is changed since it binds to the new class variable and not to the base element!

5.1.32.4 **Reason for change**

A class variable implements state that is shared by all instances of a class, and hence is global to those instances.

5.1.33 **Delete class variable**

5.1.33.1 **Description**

Delete an existing class variable from the class. Note that for this to be an atomic change, the variable should either not be used by any method, or there should be another variable of the same name that becomes visible due to this change (see discussion below).
5.1.33.2 Example

**Before change:**

```java
class Base {
    protected static int val = 1;
}
class Derived extends Base {
    protected static int val = 45;
    public static getValue() {
        return val;
    }
    // ...
}
```

**After change:**

```java
class Base {
    protected static int val = 1;
}
class Derived extends Base {
    public static getValue() {
        return val;
    }
    // ...
}
```

5.1.33.3 Equivalence

This atomic change may or may not preserve equivalence. In the example shown above, equivalence is not preserved since the LFC changes after the atomic change. Retesting is therefore necessary. Below is an example where T-equivalence is preserved.

**Before change:**

```java
class Sample {
    private static int val = 4;
    private int a = 0;
    public int ff() { return a * 9; }
}
```
After change:

class Sample {
    private static int val = 4;
    private int a = 0;
    public int ff() { return a * 9; }
}

5.1.33.4 Reason for change
The developer may introduce this atomic change as part of code restructuring.

5.1.34 Rename method

5.1.34.1 Description
Change the name of a method to something else. This is a valid atomic change only under one the following conditions:

- the method is not used anywhere
- renaming the method causes existing usage points to bind to another method with same name

5.1.34.2 Example

Before change:

class Base {
    public void ff() { /* ... */ }
}
class Derived extends Base {
    public void ff() { /* ... */ }
    public void gg() {
        ff(); // Binds to Derived.ff()
    }
}

After change:

class Base {
    public void ff() { /* ... */ }
}
class Derived extends Base {
    public void hhh() { /* ... */ }
}
5.1.34.3 Equivalence
This change may not preserve equivalence. In the above example, the behavior of Derived.gg() has changed due to the renaming of Derived.ff() to Derived.hhh(). The only time this atomic change preserves equivalence is when the method is not used anywhere.

5.1.34.4 Reason for change
The is an uncommon change.

5.1.35 Rename method argument

5.1.35.1 Description
Change the name of a formal parameter of method. This is a valid atomic change only under any of the following conditions:

- the parameter is not used within the method
- all occurrences of the old name within the method are correctly changed to the new name
- renaming the argument without changing the method body causes all occurrences of the old name within the method to bind to another object

5.1.35.2 Example
Case 1: Argument not used.

Before change:

```java
class Sample {
    private int i = 1;
    public String ff(String s, int a) {
        s = s + "hello";
        return s;
    }
}
```
After change:

class Sample {
    private int i = 1;
    public String ££(String s, int b) {
        s = s + "hello";
        return s;
    }
}

Case 2: Argument and method body are changed.

Before change:

class Sample {
    private int i = 1;
    public String ££(String s, int a) {
        s = s + a;
        return s;
    }
}

After change:

class Sample {
    private int i = 1;
    public String ££(String s, int b) {
        s = s + b;
        return s;
    }
}

Case 3: Renaming changes symbol bindings

Before change:

class Sample {
    private int i = 1;
    public String ££(String s, int a) {
        s = s + a;
        return s;
    }
}
After change:

class Sample {
    private int i = 1;
    public String ff(String s, int b) {
        s = s + i;
        return s;
    }
}

5.1.35.3 Equivalence
As can be seen from the above three cases, first two cases preserve T-equivalence, whereas the last does not.

5.1.35.4 Reason for change
This is not a common change.

5.1.36 Make method abstract

5.1.36.1 Description
Make a nonabstract method abstract. This atomic change involves the following steps:

- delete the method body
- add abstract qualifier for the method declaration
- add abstract qualifier for the class (if it is not already abstract)

Because of these steps, this atomic change is feasible only when objects of the class are not instantiated anywhere.

5.1.36.2 Example

Before change:

class A {
    public void ff() { /* ... */ }
    public void gg() { /* ... */ }
}
class B extends A {
    public void ff() { /* ... */ }
    public void gg() { /* ... */ }
}
public void hh() { /* ... */ }
}

After change:
abstract class A {
    public void ff() { /* ... */ }
    public abstract void gg();
}

class B extends A {
    public void ff() { /* ... */ }
    public void gg() { /* ... */ }
    public void hh() { /* ... */ }
}

5.1.36.3 Equivalence
Surprisingly, this atomic change preserves T-equivalence, and hence there is no need to test the class after the change (we assume as usual that the code compiles and builds correctly).

5.1.36.4 Reason for change
This atomic change is a major change in code. This change is justifiable when a substantial code restructuring takes place in the new version resulting in hierarchy alterations.

5.1.37 Make method nonabstract

5.1.37.1 Description
This atomic change involves the following steps:

- add a body for the method
- delete the abstract qualifier from its declaration
- if this is the only abstract method, consider removing the abstract qualifier from the class

5.1.37.2 Example

Before change:
abstract class A {
    public void ff() { /* ... */ }
}
```java
public abstract void gg();
}
class B extends A {
    public void ff() { /* ... */ }
    public void gg() { /* ... */ }
    public void hh() { /* ... */ }
}
```

**After change:**

```java
class A {
    public void ff() { /* ... */ }
    public void gg() { /* ... */ }
}
class B extends A {
    public void ff() { /* ... */ }
    public void gg() { /* ... */ }
    public void hh() { /* ... */ }
}
```

**5.1.37.3 Equivalence**

This atomic change preserves T-equivalence.

**5.1.37.4 Reason for change**

This atomic change is a major change in code. This change is justifiable when a substantial code restructuring takes place in the new version resulting in hierarchy alterations.

**5.1.38 Make method native**

**5.1.38.1 Description**

Turn a non-native method into native. This atomic change involves four steps:

- delete the body of the method
- qualify the method as `native`
- include the appropriate native library to be loaded into JVM in some static block
- implement the native method in C/C++
5.1.38.2 Example

**Before change:**

class Directory {
    public String[] listFiles() { /* ... */ }
}

**After change:**

class Directory {
    static {
        System.loadLibrary("mydir");
    }
    public native String[] listFiles();
}

5.1.38.3 Equivalence

This does not guarantee equivalence since, logically, the method body has been changed. Retesting is definitely necessary.

5.1.38.4 Reason for change

Making a method native is typically done because execution speed in Java is inadequate.

5.1.39 Make method non-native

5.1.39.1 Description

Turn a native method into non-native. This involves the following steps:

- define the method body in Java
- remove the native qualifier from method declaration
- remove any native implementation for the method in C/C++
- exclude the corresponding shared library from static loading into JVM
5.1.39.2 Example

**Before change:**

```java
class Directory {
    static {
        System.loadLibrary("mydir");
    }
    public native String[] listFiles();
}
```

**After change:**

```java
class Directory {
    public String[] listFiles() { /* ... */ }
}
```

5.1.39.3 Equivalence

This atomic change does not guarantee equivalence since the method implementation has changed.

5.1.39.4 Reason for change

A native method may be reimplemented in Java at a later stage taking into account machine independence and byte code portability requirements.

5.1.40 Implement an interface

5.1.40.1 Description

Add an interface to the list of interfaces this class implements. This is a valid atomic change only when the interface does not declare any method, as otherwise those methods will have to be implemented by this class, or the class will have to declared abstract.

5.1.40.2 Example

**Before change:**

```java
class MyClass {
    // ...
}
```
After change:
```java
class MyClass implements Cloneable {
    // Details
}
```

5.1.40.3 Equivalence

Adding an interface, even one that does not declare methods, can subtly change the behavior of the class. For example, in the case shown below, MyClass becomes cloneable after the change, whereas originally, it was not. It is therefore advisable to retest the class after change.

The following is an example of a program that changes in behavior due to this atomic change:
```java
class Tester {
    public static void main(String[] args) {
        Class cl = MyClass.class;
        Class[] intrf = cl.getInterfaces();
        if (intrf.length == 0)
            System.out.println("Implements none");
        else
            System.out.println("Implements "+ intrf.length + " interfaces, the first is " + intrf[0].getName());
    }
}
```

5.1.40.4 Reason for change

Interfaces such as Cloneable and Serializable are fairly useful in JDK. The designer may take advantage of these in a new version of the class.

5.1.41 Remove an interface

5.1.41.1 Description

If a class implements one or more interfaces, remove one of them from the list. If the interface declares methods that are implemented by the class, this atomic change requires that the methods be left as
they are since removing methods from a class is a different atomic change.

5.1.41.2 Example

Case 1: Interface declares no methods

**Before change:**

```java
interface Cloneable {}
class A implements Cloneable {
    // Body
}
```

**After change:**

```java
class A {
    // Body
}
```

Case 2: Interface declares methods

**Before change:**

```java
interface Runnable {
    public void run();
}
class A implements Runnable {
    public void run() {
        // ...
    }
}
```

**After change:**

```java
class A {
    public void run() {
        // ...
    }
}
```

5.1.41.3 Equivalence

This atomic change may not preserve equivalence. Hence it is required to retest the class after change.
5.1.41.4 Reason for change
Removing an interface from a class specification is not common.

5.1.42 Rename instance variable

5.1.42.1 Description
Change the name of an instance variable. For this to be a feasible atomic change, any of the following conditions must apply:

- the instance variable is not used anywhere
- renaming the variable causes current symbol bindings to change

5.1.42.2 Example
Case 1: The instance variable is unused

Before change:
class A {
    private int i;
    private double d = 12.34;
    public void f() { d += 3.4; }
}

After change:
class A {
    private int ii;
    private double d = 12.34;
    public void f() { d += 3.4; }
}

Case 2: Symbol bindings change due to renaming

Before change:
class Base {
    protected int i;
    // ...
}
class Derived extends Base {
    private int i;
    private double d = 12.34;
After change:

```java
public void f() {
    d += 3.4;
    i++;
}
```  

```java
After change:

class Base {
    protected int i;
    // ...
}
class Derived extends Base {
    private int ii;
    private double d = 12.34;
    public void f() {
        d += 3.4;
        i++;
    }
}
```

5.1.42.3 Equivalence

This atomic change preserves equivalence under Case 1, but not under Case 2 as shown above. Hence retesting may be required in these cases because it fails to preserve T-equivalence.

5.1.42.4 Reason for change

Renaming an instance variable can be required when the developer follows certain coding guidelines.

5.1.43 Rename class variable

5.1.43.1 Description

Change the name of a class variable. For this to be a feasible atomic change, any of the following conditions must apply:

- the class variable is not used anywhere
- renaming the variable causes current symbol bindings to change
5.1.43.2 Example

Case 1: The class variable is unused

**Before change:**

class A {
    private static int i;
    private double d = 12.34;
    public void f() { d += 3.4; }
}

**After change:**

class A {
    private static int ii;
    private double d = 12.34;
    public void f() { d += 3.4; }
}

Case 2: Symbol bindings change due to renaming

**Before change:**

class Base {
    protected static int i;
    // ...
}
class Derived extends Base {
    private static int i;
    private double d = 12.34;
    public void f() {
        d += 3.4;
        i++;
    }
}

**After change:**

class Base {
    protected static int i;
    // ...
}
class Derived extends Base {
    private static int ii;
private double d = 12.34;
public void f() {
    d += 3.4;
    i++;
}

5.1.43 Equivalence
This atomic change preserves equivalence under Case 1, but not under Case 2 as shown above. Hence retesting may be required in these cases because it fails to preserve T-equivalence.

5.1.43.4 Reason for change
Renaming a class variable can be required when the developer follows coding guidelines.

5.1.44 Change class variable type

5.1.44.1 Description
Change type of a class variable.

5.1.44.2 Example

Before change:

class RefCount {
    public static int count = 0;
    public RefCount() { count++; }
    protected void finalize() { count--; }
}

After change:

class RefCount {
    public static short count = 0;
    public RefCount() { count++; }
    protected void finalize() { count--; }
}

5.1.44.3 Equivalence
In general, there is no guarantee that the behavior will be the same after this atomic change. Hence this change does not necessarily preserve equivalence and retesting is called for.
5.1.44.4 Reason for change
The choice of type for a class variable is based on the set of values (domain) the variable needs to represent. For example, in the case of the RefCount class shown above, the designer feels 32 bits are not needed for representing count. So he chooses to use short (which is only 16 bits).

5.1.45 Turn instance variable into class variable

5.1.45.1 Description
An instance variable is changed into a class variable by using the qualifier static.

5.1.45.2 Example

**Before change:**

```java
class Sample {
    private int count = 0;
    // Other methods
}
```

**After change:**

```java
class Sample {
    private static int count = 0;
    // Other methods
}
```

5.1.45.3 Equivalence
This atomic change does not preserve equivalence in the general case, and hence retesting is necessary. The only time it preserves T-equivalence is when the variable in question has not been used anywhere. For instance, the following class does not require retesting after change:

**Before change:**

```java
class Sample {
    private int count = 0;
    int val = 0;
}
```
void ff() { val++; }
}

After change:
class Sample {
  private static int count = 0;
  int val = 0;
  void ff() { val++; }
}

5.1.45.4 Reason for change
This is a nontrivial change. It is conceivable that a developer who realizes a design mistake in the earlier implementation might apply this change in the next version to fix the bug. As an example, if a class is designed to keep track of the number of instantiated objects of its type, an incorrect version would be:
class RefCount {
  public int count = 0;
  public RefCount() { count++; }
  protected void finalize() { count--; }
}

The above implementation is incorrect because the count variable must be shareable across all instances, whereas it is not. If this bug is corrected, the class becomes:
class RefCount {
  public static int count = 0;
  public RefCount() { count++; }
  protected void finalize() { count--; }
}

5.1.46 Turn class variable into instance variable
5.1.46.1 Description
Change a class variable into an instance variable by removing the qualifier static. This is a valid atomic change only when the methods that use the variable in question are instance methods. If the variable is used in a static method, changing the variable to
nonstatic will generate compile time errors in the method. The reason is that using an instance variable requires this, but a static method does not have this.

5.1.46.2 Example

**Before change:**

```java
class Sample {
    private static int count = 0;
    // Other methods
}
```

**After change:**

```java
class Sample {
    private int count = 0;
    // Other methods
}
```

5.1.46.3 Equivalence

This atomic change does not preserve equivalence in the general case, and hence retesting is necessary. The only time it preserves T-equivalence is when the variable in question has not been used anywhere. For instance, the following class does not require retesting after change:

**Before change:**

```java
class Sample {
    private static int count = 0;
    int val = 0;
    void ff() { val++; }
}
```

**After change:**

```java
class Sample {
    private int count = 0;
    int val = 0;
    void ff() { val++; }
}
5.1.46.4 Reason for change
This is a nontrivial change. This change is applied usually to fix a bug in the previous implementation of a class.

5.1.47 Change method throws list

5.1.47.1 Description
Change the list of exceptions thrown from a method. In Java, a method that directly or indirectly raises checked exceptions without handling them within its body, is expected to list those exceptions in its throws list. Not doing so is a compile-time error. This rule does not apply to unchecked exceptions. Consider the class shown below:

```java
class FileTest {
    DataOutputStream dos;
    public void open(String nm) {
        try {
            dos = new DataOutputStream( new FileOutputStream(nm) );
        } catch (IOException err) {
            // Handle the exception
        }
    }
}
```

In the above example, since the open method handles the IOException (a checked exception), it need not (should not) list it in a throws clause. The above class could have been alternatively written as

```java
class FileTest {
    DataOutputStream dos;
    public void open(String nm) throws IOException{
        dos = new DataOutputStream( new FileOutputStream(nm) );
    }
}
```

In this case, if the throws list did not mention IOException, it would have been a compile-time error. On the other hand,
unchecked exceptions are not bound by this rule. The following is an example:

class ArrayCheck {
    int ar[] = new int[1];
    public int getElement(int index) {
        return ar[index]; // Array limit may be exceeded
    }
}

Accessing an element outside of an array length results in an exception called `ArrayIndexOutOfBoundsException`. However, since this is not a checked exception, it need not be handled or explicitly listed in the throws clause. The following is another possibility:

class ArrayCheck {
    int ar[] = new int[1];
    public int getElement(int index) throws
        ArrayIndexOutOfBoundsException {
        return ar[index]; // Array limit may be exceeded
    }
}

The following implementation handles the unchecked exception – another possibility:

class ArrayCheck {
    int ar[] = new int[1];
    public int getElement(int ind) {
        try {
            return ar[ind];
        } catch (ArrayIndexOutOfBoundsException e) {
            return -1;
        }
    }
    
    public static void main(String args[]) {
        System.out.println(new Test().getElement(1));
    }
}
From the discussion above, it transpires that the only time this is a feasible atomic change is when the exception being added to or deleted from throws list corresponds to the unchecked type. The assumption here is that the method body is not changed.

5.1.47.2 Example

**Before change:**

```java
class ArrayCheck {
    int ar[] = new int[1];
    public int getElement(int index) {
        return ar[index]; // Array limit may be exceeded
    }
}
```

**After change:**

```java
class ArrayCheck {
    int ar[] = new int[1];
    public int getElement(int index) throws ArrayIndexOutOfBoundsException {
        return ar[index]; // Array limit may be exceeded
    }
}
```

5.1.47.3 Equivalence

Since this atomic change deals with unchecked exceptions that are considered special, it is better to retest the class after change.

5.1.47.4 Reason for change

Merely changing a method's throws list without changing the body is not common in Java, unless it corresponds to unchecked exceptions.

5.1.48 Add init block

5.1.48.1 Description

Add an instance initializer block within the class. An instance initializer is executed every time an object of the class is instantiated.
5.1.48.2 Example

Before change:

class A {
   private int val;
   private int ff() throws MyException {
       // ...
   }
   // Other methods
}

After change:

class A {
   private int val;
   
   try {
       val = ff();
   } catch (MyException m) {
       val = 67;
   }

   private int ff() throws MyException {
       // ...
   }
   // Other methods
}

5.1.48.3 Equivalence

In general, it is not possible guarantee that adding the init block preserves equivalence. Hence retesting is necessary. The only trivial case when T-equivalence is guaranteed is when the initializer is empty!

5.1.48.4 Reason for change

One important reason for defining an instance initializer within the class is to handle initialization expressions that may result in checked exceptions (as in the above example).
5.1.49 Remove init block

5.1.49.1 Description
Remove an initializer block defined within the class.

5.1.49.2 Example

Before change:
```java
class A {
    private int a;
    {
        a = 54;
    }
    // Other elements
}
```

After change:
```java
class A {
    private int a = 54;
    // Other elements
}
```

5.1.49.3 Equivalence
In general, it is not possible guarantee that removing the init block preserves equivalence. Hence retesting is necessary. The only trivial case when T-equivalence is guaranteed is when the initializer is empty!

5.1.49.4 Reason for changes
An instance initialization block is useful in the context of nontrivial instance initializations, particularly, involving exceptions. If the initialization is simple, then there is no need for the init block.

5.1.50 Add static init block

5.1.50.1 Description
Add a static initialization block for the class. The static init block is executed exactly once, when the class is first loaded in to the virtual machine.
5.1.50.2 Example

**Before change:**

class A {
    public static int val;
    private int ff() throws MyException {
        // ...
    }
    // Other methods
}

**After change:**

class A {
    public static int val;
    static {
        try {
            val = ff();
        } catch (MyException m) {
            val = 67;
        }
    }
    private int ff() throws MyException {
        // ...
    }
    // Other methods
}

5.1.50.3 Equivalence

In general, it is not possible guarantee that adding the static init
block preserves equivalence. Hence retesting is necessary. The only
trivial case when T-equivalence is guaranteed is when the
initializer is empty!

5.1.50.4 Reason for change

One important reason for defining a static initializer within the
class is to handle initialization expressions that may result in
checked exceptions (as in the above example).
5.1.51 Remove static init block

5.1.51.1 Description
Remove a static initializer block defined within the class.

5.1.51.2 Example

Before change:
```java
class A {
    private static int a;
    {
        a = 54;
    }
    // Other elements
}
```

After change:
```java
class A {
    private static int a = 54;
    // Other elements
}
```

5.1.51.3 Equivalence
In general, it is not possible guarantee that removing the static init block preserves equivalence. Hence retesting is necessary. The only trivial case when T-equivalence is guaranteed is when the initializer is empty!

5.1.51.4 Reason for change
A static initialization block is useful in the context of nontrivial static initializations, particularly, involving exceptions. If the initialization is simple, then there is no need for the init block.

5.1.52 Change package name

5.1.52.1 Description
Change the name of current package. Every class, in Java, is associated with a default package. However, a package name can be explicitly associated with one or more classes. Package naming
is at the file level. There can be at most one package specification within a file. All classes in that file are then supposed to belong to that package. The same package name may be specified for classes within multiple files. In logical terms, it is convenient to think of a package as an aggregation of files (or modules).

5.1.52.2 Example

**Before change:**

```java
package A;
public class X {
    // ...
}
class Y {
    // ...
}
```

**After change:**

```java
package B;
public class X {
    // ...
}
class Y {
    // ...
}
```

5.1.52.3 Equivalence

Since the assumption for all atomic changes is that the software should compile and build correctly after the atomic change, and since renaming a package does not alter the behavior of a class, this change preserves $T$-equivalence. Retesting is therefore not necessary.

5.1.52.4 Reason for change

It is common for developers to initially define classes loosely without worrying about logical packaging. Once the application is fully defined (or close to that stage), a design or code review might suggest grouping in terms of packages.
5.1.53 Add inner class

5.1.53.1 Description
Define a class (or interface) in a class (or interface).

5.1.53.2 Example

Before change:
```java
class A {
    // ...
}
class B {
    public A ff() { return new A(); }
}
```

After change:
```java
class B {
    class A {
        // ...
        public A ff() { return new A(); }
    }
}
```

5.1.53.3 Equivalence
This atomic change is not guaranteed to preserve T-equivalence in all cases. In the example given above, T-equivalence is preserved. The following is an example where equivalence is not preserved:

Before change:
```java
class A {
    // ...
}
class B extends A {
    public static A ff() {
        return new A();
    }
}
```
After change:

class A {
    // ...
}
class B extends A {
    static class A {
        // ...
    }
    public static A ff() {
        return new A();
    }
}

After change, the inner class hides the base class and hence alters the behavior of B.ff()!

5.1.53.4 Reason for change
Inner classes permit a tighter coupling of related classes. For example, a Node class can be defined as a static inner class within List.

5.1.54 Delete inner class

5.1.54.1 Description
Delete a class (or interface) defined inside another class (or interface).

5.1.54.2 Example

Before change:

class B {
    class A {
        // ...
    }
    public A ff() {
        return new A();
    }
}
After change:

class A {
    // ...
}
class B {
    public A ff() {
        return new A();
    }
}

5.1.54.3 Equivalence

In cases where the atomic change does not change symbol bindings (as in the above example), retesting is not necessary. In general, however, retesting is required.

5.1.54.4 Reason for change

Inner classes represent tight coupling with corresponding outer (enclosing) classes. If the two classes are not logically coupled, it may be a good idea to keep them independent.

5.1.55 Define a local class

5.1.55.1 Description

A local class (or interface) is a class (or interface) defined within a method. Such a class is not visible outside of the method in which it is defined.

5.1.55.2 Example

Before change:

class B {
    // ...
}
class A {
    public void ff() {
        // Use B here
    }
}

After change:

class A {
    // ...
}
class B {
    public A ff() {
        return new A();
    }
}
5.1.55.3 Equivalence

This atomic change is considered to affect a method and not a class directly, since the change is made within the body of a method. In general, we cannot know whether this preserves equivalence, hence retesting is recommended.

5.1.55.4 Reason for change

If a class is useful only in the very limited context of a method, then it is better to define it as a local class.

5.1.56 Delete inner class

5.1.56.1 Description

Delete a local class (or interface).

5.1.56.2 Example

Before change:

class A {
    public void ff() {
        class B {
            // ...
        }
        // Use B here
    }
}

After change:

class A {
    public void ff() {
        class B {
            // ...
        }
        // Use B here
    }
}
After change:

class B {
    // ...
}
class A {
    public void ff() {
        // Use B here
    }
}

5.1.56.3 Equivalence
Since this occurs in the context of a method body, equivalence may not be preserved in the general case. Retesting is, therefore, advisable.

5.1.56.4 Reason for change
If a class is found to be useful outside of a method (perhaps because several methods of the same class or several classes use it), it is better to define it as a nonlocal class.

5.1.57 Define an anonymous class

5.1.57.1 Description
An anonymous class is a class defined as part of a new expression. The class is not given a name, but is indicated as a class that derives from an existing base class or as a class that implements an existing interface.

5.1.57.2 Example

Before change:

class A {
    public void ff() { /* ... */ }
}
class A2 extends A {
    public void ff() { /* ... */ }
}
class B {
    public A gg() { return new A2(); }
}

After change:

class A {
    public void ff() { /* ... */ }
}
class B {
    public A gg() {
        return new A() {
            public void ff() { /* ... */ }
        };
    }
}

5.1.57.3 Equivalence
Since this atomic change occurs in the context of a method body, in
genneral, it is advisable to retest as T-equivalence is not guaranteed.
However, it is possible that the change preserves T-
equivalence in some special cases as in the above example.

5.1.57.4 Reason for change
Anonymous classes minimize class name proliferation in Java.

5.1.58 Delete anonymous class

5.1.58.1 Description
Delete an anonymous class from within a method body.

5.1.58.2 Example

Before change:

class A {
    public void ff() { /* ... */ }
}
class B {
    public A gg() {
        return new A() {
            public void ff() { /* ... */ }
        };
    }
}
After change:

```java
class A {
    public void ff() { /* ... */ }
}
class A2 extends A {
    public void ff() { /* ... */ }
}
class B {
    public A gg() {
        return new A2();
    }
}
```

### 5.1.58.3 Equivalence
Since this atomic change occurs in the context of a method body, in general, it is advisable to retest as T-equivalence is not guaranteed. However, it is possible that the change preserves T-equivalence in some special cases as in the above example.

### 5.1.58.4 Reason for change
If a class is widely used, it is to be defined in the outer scope with a unique name. Keeping it anonymous is of no use.